Efficient topology optimization in MATLAB using 88 lines of code

Erik Andreassen · Anders Clausen · Mattias Schevenels · Boyan S. Lazarov · Ole Sigmund

Abstract The paper presents an efficient 88 line MATLAB code for topology optimization. It has been developed using the 99 line code presented by Sigmund (2001) as a starting point. The original code has been extended by a density filter, and a considerable improvement in efficiency has been achieved, mainly by preallocating arrays and vectorizing loops. A speed improvement with a factor of 100 is obtained for a benchmark example with 7500 elements. Moreover, the length of the code has been reduced to a mere 88 lines. These improvements have been accomplished without sacrificing the readability of the code. The 88 line code can therefore be considered as a valuable successor to the 99 line code, providing a practical instrument that may help to ease the learning curve for those entering the field of topology optimization. The paper also discusses simple extensions of the basic code to include recent PDE-based and black-and-white projection filtering methods. The complete 88 line code is included as an appendix and can be downloaded from the web site `www.topopt.dtu.dk`.

Keywords Topology optimization · MATLAB · Education · Computational efficiency

1 Introduction

MATLAB is a high-level programming language that allows for the solution of numerous scientific problems with a minimum of coding effort. An example is Sigmund's 99

E. Andreassen, A. Clausen, O. Sigmund*, B.S. Lazarov
Department of Mechanical Engineering, Solid Mechanics,
Technical University of Denmark, Nils Koppels Alle, B. 404,
DK-2800 Lyngby, Denmark
*E-mail: sigmund@mek.dtu.dk

M. Schevenels
Department of Civil Engineering, K.U.Leuven,
Kasteelpark Arenberg 40, B-3001 Leuven, Belgium

line topology optimization code (Sigmund, 2001). The 99 line code is intended for educational purposes and serves as an introductory example to topology optimization for students and newcomers to the field. The use of MATLAB, with its accessible syntax, excellent debugging tools, and extensive graphics handling opportunities, allows the user to focus on the physical and mathematical background of the optimization problem without being distracted by technical implementation issues. Other examples of simple MATLAB code used to provide insight in finite element analysis or topology optimization include a finite element code for the solution of elliptic problems with mixed boundary conditions on unstructured grids (Alberty et al, 1999), a similar code for problems in linear elasticity (Alberty et al, 2002), a topology optimization code for compliant mechanism design and for heat conduction problems (Bendsøe and Sigmund, 2003), a code for Pareto-optimal tracing in topology optimization (Suresh, 2010), a discrete level-set topology optimization code (Challis, 2010), and a Scilab code for two-dimensional optimization problems based on the level set method (Allaire, 2009).

Compared to high performance programming languages such as C++ and Fortran, MATLAB is generally perceived to be far behind when it comes to computational power. This can partly be explained by (1) the fact that many users apply the same programming strategies as in Fortran or C++, such as the extensive use of `for` and `while` loops, and (2) the fact that MATLAB is relatively tolerant towards bad programming practices, such as the use of dynamically growing variable arrays. In both cases the potential of MATLAB is far from optimally utilized. Efficient use of MATLAB implies loop vectorization and memory preallocation (The MathWorks, 2010). Loop vectorization is the use of vector and matrix operations in order to avoid `for` and `while` loops. Memory preallocation means that the maximum amount of memory required for an array is

reserved a priori, hence avoiding the costly operation of reallocating memory and moving data as elements are added to the array. Loop vectorization and memory preallocation are used in combination with a number of more advanced performance improving techniques in the MILAMIN code, a MATLAB program capable of solving two-dimensional finite element problems with one million unknowns in one minute on a desktop computer (Dabrowski et al, 2008).

In the 99 line topology optimization code, the performance of several operations (such as the filtering procedure and the assembly of the finite element matrices) can be increased dramatically. Partly by properly exploiting the strengths of MATLAB (using loop vectorization and memory preallocation), partly by restructuring the program (moving portions of code out of the optimization loop so that they are only executed once), a substantial increase in efficiency has been achieved: for an example problem with 7500 elements, the total computation time has been reduced by a factor 100. In addition, the original code has been extended by the inclusion of density filtering, while reducing the length of the code to only 88 lines.

The aim of this paper is to present the 88 line code. It should be considered as a successor to the 99 line code, and it is published with the same objective: to provide an educational instrument for newcomers to the field of topology optimization. The main improvements with respect to the original code are the increased speed and the inclusion of a density filter. These are relevant improvements, as the 99 line code has been downloaded by a more than 8000 people since 1999 and is still used as a basis for new developments in the field of topology optimization. The density filter is a useful addition as it paves the way for the implementation of more modern filters such as the Heaviside filters proposed by Guest et al (2004) and Sigmund (2007).

The present text is conceived as an extension of the paper by Sigmund (2001). Large parts of the 88 line code are identical to the original 99 line code, and the same notation is adopted. This approach is followed in an attempt to minimize the effort required to upgrade to the new implementation.

The paper is organized as follows. The topology optimization problem is formulated in section 2. As in the original paper, the focus is restricted to minimum compliance problems with a constraint on the amount of material available. The 88 line code is explained in section 3. Special attention is paid to the portions of the code that have changed with respect to the original 99 line code. These two sections constitute the core of the paper. The remaining sections have a supplementary character, addressing variants of and extensions of the 88 line code and discussing its performance. Section 4 presents two alternative implementations of the filtering operation. The first alternative is based on the built-in MATLAB convolution operator function `conv2`. This modification implies a further reduction of the code to 71 lines and leads to a reduction of the memory footprint, but this comes at the expense of the code's readability for those unfamiliar with the `conv2` function. The second alternative is based on the application of a Helmholtz type partial differential equation to the density or sensitivity field (Lazarov and Sigmund, 2010). This approach allows for the use of a finite element solver to perform the filtering operation, which reduces the complexity of the implementation for serial and parallel machines, as well as the computation time for large problems and complex geometries. Section 5 shows how to extend the 88 line code to problems involving different boundary conditions, multiple load cases, and passive elements. Furthermore, the inclusion of a Heaviside filter in order to obtain black-and-white solutions is elaborated. In section 6, the performance of the 88 line code and its variants is examined. The computation time is analyzed for three benchmark examples solved with both the original 99-line code and the new versions of the code. The memory usage of the new code is also briefly discussed.

2 Problem formulation

The MBB beam is a classical problem in topology optimization. In accordance with the original paper (Sigmund, 2001), the MBB beam is used here as an example. The design domain, the boundary conditions, and the external load for the MBB beam are shown in figure 1. The aim of the optimization problem is to find the optimal material distribution, in terms of minimum compliance, with a constraint on the total amount of material.
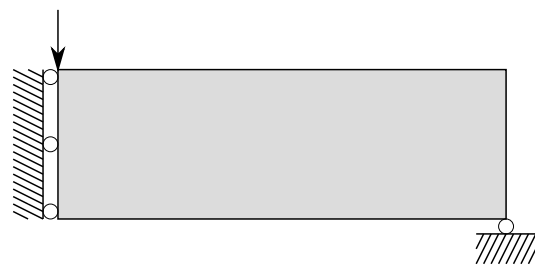

Fig. 1 The design domain, boundary conditions, and external load for the optimization of a symmetric MBB beam.

2.1 Modified SIMP approach

The design domain is discretized by square finite elements and a "density-based approach to topology optimization" is followed (Bendsøe, 1989; Zhou and Rozvany, 1991); i.e.

each element e is assigned a density x_e that determines its Young's modulus E_e:

$$E_e(x_e) = E_{\min} + x_e^p(E_0 - E_{\min}), \qquad x_e \in [0, 1] \qquad (1)$$

where E_0 is the stiffness of the material, $E_{\min}$ is a very small stiffness assigned to void regions in order to prevent the stiffness matrix from becoming singular, and p is a penalization factor (typically $p = 3$) introduced to ensure black-and-white solutions. Equation (1) corresponds to the modified SIMP approach, which differs from the classical SIMP approach used in the original paper in the occurrence of the term $E_{\min}$. In the classical SIMP approach, elements with zero stiffness are avoided by imposing a lower limit slightly larger than zero on the densities x_e. The modified SIMP approach has a number of advantages (Sigmund, 2007), most importantly that it allows for a straightforward implementation of additional filters, as illustrated in section 5.

The mathematical formulation of the optimization problem reads as follows:

$$\begin{aligned} \min_{\mathbf{x}}: \quad & c(\mathbf{x}) = \mathbf{U}^{\mathrm{T}}\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} E_e(x_e)\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_0\mathbf{u}_e \\ \text{subject to}: \quad & V(\mathbf{x})/V_0 = f \\ & \mathbf{K}\mathbf{U} = \mathbf{F} \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} \end{aligned} \qquad (2)$$

where c is the compliance, $\mathbf{U}$ and $\mathbf{F}$ are the global displacement and force vectors, respectively, $\mathbf{K}$ is the global stiffness matrix, $\mathbf{u}_e$ is the element displacement vector, $\mathbf{k}_0$ is the element stiffness matrix for an element with unit Young's modulus, $\mathbf{x}$ is the vector of design variables (i.e. the element densities), N is the number of elements used to discretize the design domain, $V(\mathbf{x})$ and V_0 are the material volume and design domain volume, respectively, and f is the prescribed volume fraction.

2.2 Optimality criteria method

The optimization problem (2) is solved by means of a standard optimality criteria method. A heuristic updating scheme identical to the scheme used in the original paper is followed:

$$x_e^{\text{new}} = \begin{cases} \max(0, x_e - m) & \text{if } x_e B_e^\eta \leq \max(0, x_e - m) \\ \min(1, x_e + m) & \text{if } x_e B_e^\eta \geq \min(1, x_e - m) \\ x_e B_e^\eta & \text{otherwise} \end{cases} \qquad (3)$$

where m is a positive move limit, $\eta \, (= 1/2)$ is a numerical damping coefficient, and B_e is obtained from the optimality

condition as:

$$B_e = \frac{-\dfrac{\partial c}{\partial x_e}}{\lambda \dfrac{\partial V}{\partial x_e}} \qquad (4)$$

where the Lagrangian multiplier λ must be chosen so that the volume constraint is satisfied; the appropriate value can be found by means of a bisection algorithm.

The sensitivities of the objective function c and the material volume V with respect to the element densities x_e are given by:

$$\frac{\partial c}{\partial x_e} = -px_e^{p-1}(E_0 - E_{\min})\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_0\mathbf{u} \qquad (5)$$

$$\frac{\partial V}{\partial x_e} = 1 \qquad (6)$$

Equation (6) is based on the assumption that each element has unit volume.

2.3 Filtering

In order to ensure existence of solutions to the topology optimization problem and to avoid the formation of checkerboard patterns (Díaz and Sigmund, 1995; Jog and Haber, 1996; Sigmund and Petersson, 1998), some restriction on the design must be imposed. A common approach is the application of a filter to either the sensitivities or the densities. A whole range of filtering methods is thoroughly described by Sigmund (2007). In addition to the sensitivity filter (Sigmund, 1994, 1997), which is already implemented in the 99 line code, the new 88 line code also includes density filtering (Bruns and Tortorelli, 2001; Bourdin, 2001).

The sensitivity filter modifies the sensitivities $\partial c/\partial x_e$ as follows:

$$\widehat{\frac{\partial c}{\partial x_e}} = \frac{1}{\max(\gamma, x_e)\displaystyle\sum_{i \in N_e} H_{ei}} \sum_{i \in N_e} H_{ei} x_i \frac{\partial c}{\partial x_i} \qquad (7)$$

where N_e is the set of elements i for which the center-to-center distance $\Delta(e, i)$ to element e is smaller than the filter radius $r_{\min}$ and H_{ei} is a weight factor defined as:

$$H_{ei} = \max\left(0, r_{\min} - \Delta(e, i)\right) \qquad (8)$$

The term $\gamma \, (= 10^{-3})$ in equation (7) is a small positive number introduced in order to avoid division by zero. This is a difference as compared to the original paper, where the classical SIMP approach is used. In the classical SIMP approach, the density variables cannot become zero, and the term γ is not required.

The density filter transforms the original densities x_e as follows:

$$\tilde{x}_e = \frac{1}{\sum\limits_{i \in N_e} H_{ei}} \sum\limits_{i \in N_e} H_{ei} x_i \qquad (9)$$

In the following, the original densities x_e are referred to as the design variables. The filtered densities $\tilde{x}_e$ are referred to as the physical densities. This terminology is used to stress the fact that the application of a density filter causes the original densities x_e to loose their physical meaning. One should therefore always present the filtered density field $\tilde{x}_e$ rather than the original density field x_e as the solution to the optimization problem (Sigmund, 2007).

In the case where a density filter is applied, the sensitivities of the objective function c and the material volume V with respect to the physical densities $\tilde{x}_e$ are still given by equations (5) and (6), provided that the variable x_e is replaced with $\tilde{x}_e$. The sensitivities with respect to the design variables x_j are obtained by means of the chain rule:

$$\frac{\partial \psi}{\partial x_j} = \sum_{e \in N_j} \frac{\partial \psi}{\partial \tilde{x}_e} \frac{\partial \tilde{x}_e}{\partial x_j} = \sum_{e \in N_j} \frac{1}{\sum\limits_{i \in N_e} H_{ei}} H_{je} \frac{\partial \psi}{\partial \tilde{x}_e} \qquad (10)$$

where the function ψ represents either the objective function c or the material volume V.

3 MATLAB implementation

In this section the 88 line MATLAB code (see appendix) is explained. The code is called from the MATLAB prompt by means of the following line:

```
top88(nelx,nely,volfrac,penal,rmin,ft)
```

where `nelx` and `nely` are the number of elements in the horizontal and vertical direction, respectively, `volfrac` is the prescribed volume fraction f, `penal` is the penalization power p, `rmin` is the filter radius r_{min} (divided by the element size), and the additional argument (compared to the 99 line code) `ft` specifies whether sensitivity filtering (`ft = 1`) or density filtering (`ft = 2`) should be used. When sensitivity filtering is chosen, the 88 line code yields practically[1] the same results as the 99 line code; e.g. the optimized MBB beam shown in figure 1 of the original paper by Sigmund (2001) can be reproduced by means of the following function call:

```
top88(60,20,0.5,3,1.5,1)
```

The most obvious differences between the 88 line code and the 99 line code are the following: (1) the `for` loops

used to assemble the finite element matrices, to compute the compliance, and to perform the filtering operation have been vectorized, (2) the remaining arrays constructed by means of a `for` loop are properly preallocated, (3) a maximum amount of code is moved out of the optimization loop to ensure that it is only executed once, (4) a distinction is made between the design variables `x` and the physical densities `xPhys` in order to facilitate the application of a density filter, and (5) all subroutines have been integrated in the main program.

The 88 line code consists of three parts: the finite element analysis, the sensitivity or density filter, and the optimization loop. These parts are discussed in detail in subsections 3.1 to 3.3. Subsection 3.4 presents some results obtained with the 88 line code.

3.1 Finite element analysis

The design domain is assumed to be rectangular and discretized with square elements. A coarse example mesh consisting of 12 elements with four nodes per element and two degrees of freedom (DOFs) per node is presented in figure 2. Both nodes and elements are numbered column-wise from left to right, and the DOFs $2n - 1$ and $2n$ correspond to the horizontal and vertical displacement of node n, respectively. This highly regular mesh can be exploited in several ways in order to reduce the computational effort in the optimization loop to a minimum.
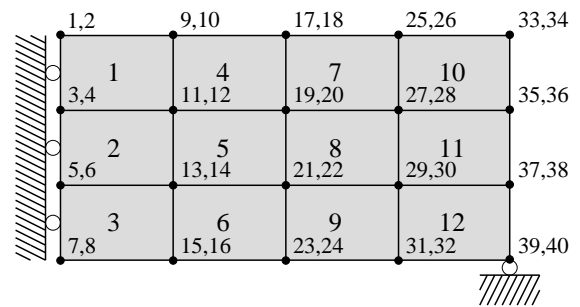

Fig. 2 The design domain with 12 elements.

The finite element preprocessing part starts with the definition of the material properties (lines 4-6): E0 is the Young's modulus E_0 of the material, `Emin` is the artificial Young's modulus E_{min} assigned to void regions (or the Young's modulus of the second material in a two-phase design problem), and `nu` is the Poisson's ratio ν.

Next the element stiffness matrix $\mathbf{k}_0$ for an element with unit Young's modulus is computed (lines 8-12). This matrix is denoted as KE. Due to the regularity of the mesh, this matrix is identical for all elements.

---

[1] The slight difference which can be observed between the 88-line and the 99-line code is due to the difference in the SIMP formulation.

In order to allow for an efficient assembly of the stiffness matrix in the optimization loop, a matrix `edofMat` is constructed (lines 13-15). The i-th row of this matrix contains the eight DOF indices corresponding to the i-th element (in a similar way as the `edof` vector in the original 99 line code). The matrix `edofMat` is constructed in three steps. First, a $(\text{nely} + 1) \times (\text{nelx} + 1)$ matrix `nodenrs` with the node numbers is defined. The MATLAB function `reshape` is used; this function returns a matrix with the size specified by the second and third input argument, whose elements are taken column-wise from the first input argument (which is in this case a vector containing the node numbers). Next, the matrix `nodenrs` is used to determine the first DOF index for all elements, which are stored in a vector `edofVec`. Finally, the matrix `edofVec` is used to determine the eight DOF indices for each element. To this end, the MATLAB function `repmat` is called twice. This function copies a matrix the specified number of times in the vertical and horizontal direction. The first call to the `repmat` function returns a matrix with eight columns which are all copies of the vector `edofVec`. The second call returns a matrix of the same size where all rows are identical; this matrix relates the indices of the eight DOFs of an element to the index of its first DOF stored in the vector `edofVec`. The results are added up and collected in the matrix `edofMat`. For the example mesh shown in figure 2, this procedure yields the following result:

$$\text{edofMat} = \begin{bmatrix} 3 & 4 & 11 & 12 & 9 & 10 & 1 & 2 \\ 5 & 6 & 13 & 14 & 11 & 12 & 3 & 4 \\ 7 & 8 & 15 & 16 & 13 & 14 & 5 & 6 \\ 11 & 12 & 19 & 20 & 17 & 18 & 9 & 10 \\ \vdots & \vdots & \vdots & \vdots & & & & \\ 31 & 32 & 39 & 40 & 37 & 38 & 29 & 30 \end{bmatrix} \begin{matrix} \leftarrow \text{Element 1} \\ \leftarrow \text{Element 2} \\ \leftarrow \text{Element 3} \\ \leftarrow \text{Element 4} \\ \\ \leftarrow \text{Element 12} \end{matrix}$$

In each iteration of the optimization loop, the assembly of the global stiffness matrix $\mathbf{K}$ is efficiently performed by means of the `sparse` function in MATLAB, so avoiding the use of `for` loops. The procedure followed here is inspired by the approach described by Davis (2007). The `sparse` function takes three vectors as input arguments: the first and second contain the row and column indices of the non-zero matrix entries, which are collected in the third vector. Specifying the same row and column indices multiple times results in a summation of the corresponding entries.

The row and colums index vectors (`iK` and `jK`, respectively) are created in lines 16-17 using the `edofMat` matrix. Use is made of a Kronecker matrix product with a unit vector of length 8, followed by a reshaping operation. The resulting vectors `iK` and `jK` are structured so that the indices $\text{iK}(k)$ and $\text{jK}(k)$ correspond to the (i, j)-th entry of the stiffness matrix for element e, where $k = i + 8(j - 1) + 64(e - 1)$.

The third vector, containing the entries of the sparse stiffness matrix, is computed in the optimization loop (line 54), as it depends on the physical densities x̃. This vector `sK` is obtained by reshaping the element stiffness matrix `KE` to obtain a column vector, multiplying this vector with the appropriate Young's modulus $E_e(\tilde{x}_e)$ for each element, and concatenating the results for all elements. The multiplication and concatenation are implemented as a matrix product followed by a reshaping operation.

The actual assembly of the stiffness matrix `K` is performed on line 55 by means of the `sparse` function, using the index vectors `iK` and `jK` and the vector with non-zero entries `sK`. This procedure could be further improved by using the `sparse2` function from CHOLMOD (Davis, 2008), which is faster than the standard MATLAB `sparse` function due to the use of a more efficient sorting algorithm for the indices, but this is beyond the scope of the present paper. The second statement on line 55 ensures that the stiffness matrix is perfectly symmetric. This is important as it determines the algorithm used by MATLAB to solve the system of finite element equations. If the stiffness matrix is sparse, symmetric, and has real positive diagonal elements, Cholesky factorization is used. If the stiffness matrix is not symmetric (due to rounding errors in the assembly procedure), LU factorization is used instead, resulting in a longer computation time.

The boundary conditions and the load vector are defined on lines 18-23. These lines are almost identical to those in the original 99 line code and are therefore not discussed in the present paper. The main difference with the original code is that these lines are moved out of the optimization loop.

The system of finite element equations is finally solved on line 56.

3.2 Filtering

The application of a sensitivity filter according to equation (7) involves a weighted average over different elements. This is a linear operation; it can therefore be implemented as a matrix product of a coefficient matrix and a vector containing the original sensitivities $\partial c / \partial x_i$ (multiplied with the design variables x_i). Dividing the result by a factor $\max(\gamma, x_e) \sum_{i \in N_e} H_{ei}$ yields the filtered sensitivities $\widehat{\partial c / \partial x_e}$. This operation is performed on line 64. The matrix `H` and the vector `Hs` contain the coefficients H_{ei} and the normalization constants $\sum_{i \in N_e} H_{ei}$, respectively.

The use of a density filter not only implies filtering of the densities according to equation (9) but also a chain rule modification of the sensitivities of the objective function and the volume constraint according to equation (10). Both operations involve a weighted average over different elements. The density filtering is performed on line 77, the

modification of the sensitivities on lines 66-67. Use is made of the same coefficients H and normalization constants Hs as described above.

Both the matrix H and the vector Hs remain invariant during the optimization and are computed a priori. The (nelx × nely) × (nelx × nely) coefficient matrix H establishes a relationship between all elements. However, as the filter kernel defined in equation (8) has a bounded support, only neighboring elements affect one another. As a consequence, the majority of the coefficients is zero and the matrix H is sparse. It is constructed by means of the built-in sparse MATLAB function. Row and column index vectors iH and jH as well as a vector sH with non-zero entries are assembled by means of four nested for loops on lines 25-42. In order to avoid continuous resizing of these vectors as entries are added, a sufficient (but slightly too high) amount of memory is preallocated. The entries that remain unused in the vectors iH, jH, and sH have no effect: they preserve their initial value (1, 1, and 0, respectively) and result in the addition of a zero term to the first element of the sparse matrix H. The assembly of the matrix H from the vectors iH, jH, and sH is performed on line 43. The vector Hs is subsequently computed on line 44.

3.3 Optimization loop

The main part of the 88 line code is the optimization loop. The loop is initialized on lines 46-49. All design variables x_e are initially set equal to the prescribed volume fraction f. The corresponding physical densities $\tilde{x}_e$ are identical to the design variables x_e: in the sensitivity filtering approach, this equality always holds, while in the density filtering approach, it holds as long as the design variables represent a homogeneous field. For other types of filters (especially non-volume-preserving filters), it may be necessary to compute the initial physical densities $\tilde{x}_e$ by explicit application of the filter to the initial design variables x_e, and to adjust the initial design variables in such a way that the volume constraint is satisfied (as this constraint is specified in terms of the physical densities $\tilde{x}_e$).

Each iteration of the optimization loop starts with the finite element analysis as described in subsection 3.1 (lines 54-56).

Next, the objective function (the compliance) c is computed, as well as the sensitivities dc and dv of the objective function and the volume constraint with respect to the physical densities (lines 58-61). Compared to the original 99 line code, efficient use is made of the edofMat matrix to compute the compliance for all elements simultaneously: the edofMat matrix is used as an index into the displacement vector U, resulting in a matrix with the size of edofMat that contains the displacements corresponding to the DOFs listed in edofMat.

The sensitivities are subsequently filtered (if sensitivity filtering is used) or modified (if density filtering is used) as explained in subsection 3.2 (lines 63-68).

On lines 70-82, an optimality criteria method is used to update the design variables according to equation (3). The update is performed in a similar way as in the original 99 line code, except that (1) the sensitivity dv of the volume constraint is explicitly taken into account, (2) the Lagrange multiplier lmid is determined using the physical densities instead of the design variables, and (3) the stop condition is specified in relative terms. The first change is made for the sake of the density filter: in the sensitivity filtering approach, the sensitivities dv are identical for all elements and can therefore be omitted from the definition of the heuristic updating factor B_e, but in the density filtering approach, this is no longer true due to the modification of the sensitivities performed on line 67. The second change is strictly speaking not necessary: the density filter is volume-preserving, which means that the volume constraint can equally well be evaluated in terms of the design variables. When another (non-volume-preserving) filter is applied, however, it is absolutely necessary to evaluate the volume constraint in terms of the physical densities. The third change is simply made to optimize the balance between accuracy and computation speed.

Finally, the intermediate results are printed (lines 84-85) and plotted (line 87) in the same way as in the original 99 line code.

The optimization loop is terminated when the L_∞ norm of the difference between two consecutive designs (in terms of design variables) is less than 1 percent.

3.4 Results

The 88 line code is used to optimize the MBB beam. Three different mesh sizes are considered, consisting of 60×20 elements, 150×50 elements, and 300×100 elements. The volume constraint is set to $50\,\%$ and the usual value $p = 3$ is used for the penalization exponent. The problem is solved using sensitivity and density filtering. The filter radius $r_{\min}$ equals 0.04 times the width of the design domain, i.e. 2.4, 6, and 16 for the different meshes.

Figure 3 shows the optimized design and the corresponding compliance c. The figures demonstrate that both sensitivity filtering and density filtering suppress checkerboard patterns and lead to mesh independent designs; refining the mesh only leads to a refinement of the solution, not to a different topology.
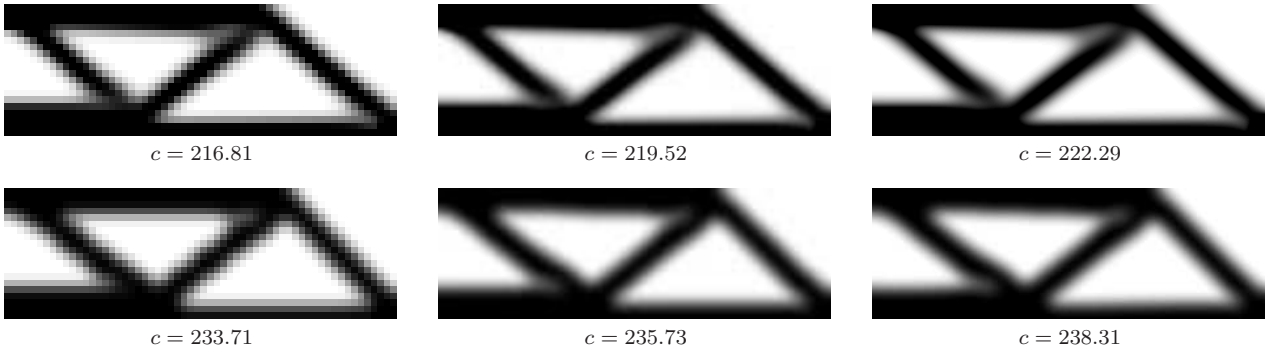
$c = 216.81$ $c = 219.52$ $c = 222.29$

$c = 233.71$ $c = 235.73$ $c = 238.31$

Fig. 3 Optimized design of the MBB beam and corresponding compliance c obtained with the 88 line code using sensitivity filtering (top) and density filtering (bottom). A mesh with 60×20 elements (left), 150×50 elements (middle), and 300×100 elements (right) has been used.

4 Alternative implementations

This section presents two alternatives to the 88 line code discussed in the previous section. The focus is on the implementation of the filters.

The first alternative makes use of the built-in MATLAB function `conv2`. This approach is mathematically equivalent to the implementation presented in the previous section, and it allows for a reduction of the code to 71 lines. It also leads to a reduction of the memory requirements, as will be discussed in subsection 6.2. A possible disadvantage of this approach is that it may obfuscate the filtering procedure for readers unfamiliar with the `conv2` function and that its applicability is limited to regular meshes.

The second alternative presents the use of filtering based on a Helmholtz type partial differential equation (PDE). This approach allows for the use of a finite element solver to perform the filtering operation, which speeds up significantly the filtering process for three-dimensional problems and simplifies parallel implementations of filtered topology optimization problems. The results obtained with the PDE filter are similar to those obtained using an exponentially decaying filter kernel (Bruns and Tortorelli, 2001).

4.1 Filtering using the CONV2 function

The optimization problem discussed in the previous sections has two properties that allow for a more concise implementation. First, a rectangular mesh consisting of rectangular (square) finite elements is used. Second, the filter kernel is invariant in space (or, loosely speaking, the filter radius $r_{\min}$ is the same at all positions in the design domain). As a consequence, the filtering operation can be interpreted as a two-dimensional discrete convolution. In the following paragraphs, the convolution based approach is elaborated for the filtering of the densities. The filtering or modification of the sensitivities can be addressed in a similar way.

The density filter, defined in equation (9), is reformulated as follows:

$$\tilde{x}_{(k,l)} = \frac{\sum_{m,n} \mathcal{H}(m,n) x_{(k-m,l-n)}}{\sum_{m,n} \mathcal{H}(m,n)} \tag{11}$$

where $x_{(i,j)}$ and $\tilde{x}_{(i,j)}$ denote the design variable and the physical density, respectively, for the element in the i-th row and the j-th column. The filter kernel $\mathcal{H}(m,n)$ is a function of the discrete variables m and n:

$$\mathcal{H}(m,n) = \max\left(0, r_{\min} - \delta(m,n)\right) \tag{12}$$

where $\delta(m,n)$ represents the center-to-center distance between two elements separated by m rows and n columns. Both sums in equation (11) must be taken over all indices m and n for which the kernel $\mathcal{H}(m,n)$ is non-zero and for which $(k-m, l-n)$ refers to an existing element.

The non-zero part of the filter kernel $\mathcal{H}(m,n)$ can be expressed as an $M \times N$ matrix $\mathbf{h}$ defined as:

$$h_{\left(m+\frac{M+1}{2}, n+\frac{N+1}{2}\right)} = \mathcal{H}(m,n) \tag{13}$$

Introducing equation (13) in equation (11) yields the following expression:

$$\tilde{x}_{(k,l)} = \frac{\sum_{m,n} h_{\left(m+\frac{M+1}{2}, n+\frac{N+1}{2}\right)} x_{(k-m,l-n)}}{\sum_{m,n} h_{\left(m+\frac{M+1}{2}, n+\frac{N+1}{2}\right)}} \tag{14}$$

The sum in the numerator corresponds to the (k,l)-th element of the central part of the two-dimensional convolution of the matrices $\mathbf{x}$ and $\mathbf{h}$, which is obtained in MATLAB as `conv2(x,h,'same')`. The sum in the denominator must be taken over the same indices, which is most easily accomplished by using the same MATLAB code as for the numerator, substituting the matrix $\mathbf{x}$ with a unit matrix of the same size.

Using the convolution based approach for the density filter, the modification of the sensitivities, and the sensitivity filter allows for a reduction of the 88 line code to 71 lines. Three modifications are required.

First, the preparation of the filter (lines 25-44) is replaced with the following lines:

```
[dy,dx] = meshgrid(-ceil(rmin)+1:ceil(rmin)-1, ...
                    -ceil(rmin)+1:ceil(rmin)-1);
h = max(0,rmin-sqrt(dx.^2+dy.^2));
Hs = conv2(ones(nely,nelx),h,'same');
```

where the matrix h is the non-zero part of the filter kernel and the matrix Hs represents the sum in the denominator on the right hand side of equation (14). This sum does not change during the optimization loop and is therefore computed in advance. Note that the matrix Hs computed here is identical to the matrix Hs in the 88 line code.

Second, the filtering or modification of the sensitivities (lines 63-68) is replaced with the following code:

```
if ft == 1
  dc = conv2(dc.*xPhys,h,'same')./Hs./max(1e-3,xPhys);
elseif ft == 2
  dc = conv2(dc./Hs,h,'same');
  dv = conv2(dv./Hs,h,'same');
end
```

Third, the filtering of the densities (line 77) is replaced with the following line:

```
xPhys = conv2(xnew,h,'same')./Hs;
```

4.2 Filtering based on Helmholtz type differential equations

The density filter given by (9) can be implicitly represented by the solution of a Helmholtz type PDE (Lazarov and Sigmund, 2010) with homogeneous Neumann boundary conditions:

$$-R_{\min}^2 \nabla^2 \tilde{\psi} + \tilde{\psi} = \psi \tag{15}$$

$$\frac{\partial \tilde{\psi}}{\partial \mathbf{n}} = 0 \tag{16}$$

where ψ is a continuous representation of the unfiltered design field, and $\tilde{\psi}$ is the filtered field. The solution of the above PDE can be written in the form of a convolution integral which is equivalent to the classical filter. The parameter $R_{\min}$ in (15) plays a similar role as $r_{\min}$ in (8). An approximate relation between the length scales for the classical and the PDE filter is given by (Lazarov and Sigmund, 2010):

$$R_{\min} = r_{\min}/2\sqrt{3} \tag{17}$$

The PDE filter is volume preserving, i.e. the volume of the input field is equal to the volume of the filtered field. The same idea can be applied as a sensitivity filter with the input

field in (15) replaced by $\psi = x\frac{\partial c}{\partial x}$ and the output field given by $\tilde{\psi} = x\frac{\partial c}{\partial x}$ (Lazarov and Sigmund, 2009).

The filter properties have been discussed extensively by Lazarov and Sigmund (2010), and here only the main advantages with respect to memory usage and computational cost are highlighted. The classical filter requires information about the neighbor elements, which for irregular meshes and complex geometries is obtained by a relatively expensive search. Clearly the approach presented in subsection 3.2 speeds up the filtering process if the search procedure is performed only once as a preprocessing step, however, the computational complexity and the memory utilization are proportional to $r_{\min}^2$ in two dimensions and to $r_{\min}^3$ in three dimensions, respectively. The PDE filter approach utilizes the mesh used for the state problem and does not require any additional information, which avoids excessive memory usage. Furthermore, the computational cost depends linearly on the length parameter $R_{\min}$ if the solution of the PDE (15) is obtained by an iterative method. Therefore, for a large filter radius, especially in three dimensions, the PDE filtering scheme should be the preferred choice. In the presented two-dimensional examples with a regular mesh, the concept will not result in improved performance, however, we include it here for educational reasons and inspiration.

FE discretization of equation (15) leads to the following system of linear equations:

$$\mathbf{K}_F \tilde{\mathbf{x}}_N = \mathbf{T}_F \mathbf{x} \tag{18}$$

where $\mathbf{K}_F$ is the standard FE stiffness matrix for scalar problems, $\mathbf{T}_F$ is a matrix which maps the element design values $\mathbf{x}$ to a vector with nodal values, and $\tilde{\mathbf{x}}_N$ is the nodal representation of the filtered field. The element-wise representation of the filtered field is obtained as:

$$\tilde{\mathbf{x}} = \mathbf{T}_F^T \tilde{\mathbf{x}}_N \tag{19}$$

The PDE filter requires minor changes of the 88 line code and reduces it to 82 lines. The preparation of the filter (lines 25-44) is replaced with the following lines:

```
Rmin = rmin/2/sqrt(3);
KEF = Rmin^2*[4 -1 -2 -1; -1  4 -1 -2; ...
             -2 -1  4 -1; -1 -2 -1  4]/6 + ...
             [4 2 1 2; 2 4 2 1; ...
              1 2 4 2; 2 1 2 4]/36;
edofVecF = reshape(nodenrs(1:end-1,1:end-1),nelx*nely,1);
edofMatF = repmat(edofVecF,1,4) + ...
           repmat([0 nely+[1:2] 1],nelx*nely,1);
iKF = reshape(kron(edofMatF,ones(4,1))',16*nelx*nely,1);
jKF = reshape(kron(edofMatF,ones(1,4))',16*nelx*nely,1);
sKF = reshape(KEF(:)*ones(1,nelx*nely),16*nelx*nely,1);
KF = sparse(iKF,jKF,sKF);
LF = chol(KF,'lower');
iTF = reshape(edofMatF,4*nelx*nely,1);
jTF = reshape(repmat([1:nelx*nely],4,1)',4*nelx*nely,1);
sTF = repmat(1/4,4*nelx*nely,1);
TF = sparse(iTF,jTF,sTF);
```
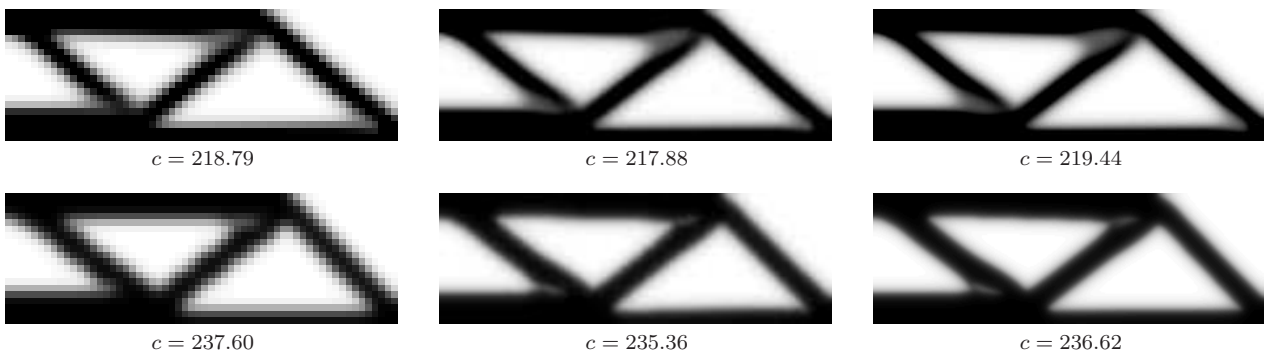
Fig. 4 Optimized design of the MBB beam and corresponding compliance c obtained with the variant of the 88 line code using PDE based sensitivity filtering (top) and density filtering (bottom). A mesh with 60×20 elements (left), 150×50 elements (middle), and 300×100 elements (right) has been used.

where KF corresponds to the tangent filter matrix and TF corresponds to the transformation matrix on the right hand side of equation (18). In order to keep the MATLAB code readable, the linear system obtained by FE discretization of equation (15) is solved by factorization instead of an iterative method, which hides some of the filter advantages. The second change is a replacement of the filtering or modification of the sensitivities (lines 63-68) with the following code:

```
if ft == 1
  dc(:) = (TF'*(LF'\(LF\(TF*(dc(:).*xPhys(:)))))) ...
    ./max(1e-3,xPhys(:));
elseif ft == 2
  dc(:) = TF'*(LF'\(LF\(TF*dc(:))));
  dv(:) = TF'*(LF'\(LF\(TF*dv(:))));
end
```

Finally, the filtering of the densities (line 77) is replaced with the following line:

```
xPhys(:) = (TF'*(LF'\(LF\(TF*xnew(:)))));
```

Figure 4 shows the optimized MBB beam and the corresponding compliance c obtained with the PDE filter, using the same input parameters as in subsection 3.4. The figure shows that the PDE filter leads to a mesh independent design without checkerboard patterns. The optimized design and the corresponding compliance c are similar to those obtained with the standard density and sensitivity filters shown in figure 3. They are not identical, however. The difference is due to the fact that the PDE filter is based on an exponentially decaying filter kernel, while the standard filters are based on a linearly decaying filter kernel.

5 Extensions

Sigmund (2001) describes how to extend the 99 line code to account for different boundary conditions, multiple load cases, and passive elements, and how to replace the optimality criteria based optimizer with a more general optimization scheme. In this section, the extensions discussed in the original paper are reconsidered, now starting from the 88 line code. In addition, the implementation of a black-and-white projection filter is also addressed.
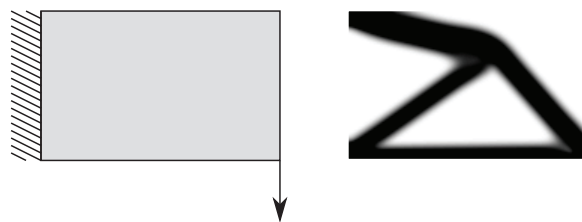
5.1 Other boundary conditions

Fig. 5 The design domain, boundary conditions, and external load for the optimization of a cantilever beam (left) and the optimized design obtained with a variant of the 88 line code using sensitivity filtering (right).

Changing load and support conditions in order to solve other optimization problems is very straightforward. In order to solve the short cantilever example shown in figure 5, line 19 of the 88 line code must be changed to:

```
F = sparse(2*(nely+1)*(nelx+1),1,-1, ...
    2*(nely+1)*(nelx+1),1);
```

Line 21 must be changed to:

```
fixeddofs = [1:2*nely+1];
```

With these changes, the optimized design shown in figure 5 is obtained by means of the following function call:

```
top88(160,100,0.4,3,6,1)
```

5.2 Multiple load cases

It is also very simple to extend the algorithm to account for multiple load cases. As an example, the problem outlined in figure 6 is considered.
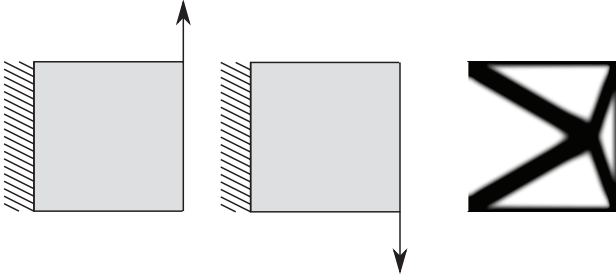
Fig. 6 The design domain, boundary conditions, and external loads for the optimization of a cantilever beam with two load cases (left and middle) and the optimized design obtained with a variant of the 88 line code using sensitivity filtering (right).

In the case of two load cases, the force and displacement vectors must be defined as two-column vectors, which means that lines 19 and 20 are changed to:

```
F = sparse([2*(nely+1)*nelx+2,2*(nely+1)*(nelx+1)], ...
    [1 2],[1 -1],2*(nely+1)*(nelx+1),2);
U = zeros(2*(nely+1)*(nelx+1),2);
```

The support conditions (line 21) are defined in the same way as in the previous subsection. The equilibrium equations must be solved for both load cases, which is accomplished by changing line 56 as follows:

```
U(freedofs,:) = K(freedofs,freedofs)\F(freedofs,:);
```

The objective function is now defined as the sum of two compliances:

$$c(\mathbf{x}) = \sum_{i=1}^{2} \mathbf{U}_i^{\mathrm{T}} \mathbf{K} \mathbf{U}_i \qquad (20)$$

Lines 58-60 are thus replaced with the following code:

```
c=0;
dc=0;
for i = 1:size(F,2)
  Ui = U(:,i);
  ce = reshape(sum((Ui(edofMat)*KE).*Ui(edofMat),2), ...
        nely,nelx);
  c = c + sum(sum((Emin+xPhys.^penal*(E0-Emin)).*ce));
  dc = dc - penal*(E0-Emin)*xPhys.^(penal-1).*ce;
end
```

The optimized design shown in figure 6 can now be obtained by means of the following function call:

```
top88(150,150,0.4,3,6,1)
```

5.3 Passive elements

In some cases, certain areas of the design domain may be required to be void or solid (e.g. to allow for the passage of a pipe or to support a secondary structure). This can be easily accomplished by means of the 88 line code through the definition of passive elements, i.e. elements with a density fixed to be zero or one. As an example, the optimization problem defined in figure 7 is addressed. A circular region of the design domain with radius $\mathtt{nely}/3$ and center $(\mathtt{nely}/2, \mathtt{nelx}/3)$ is fixed to be void.
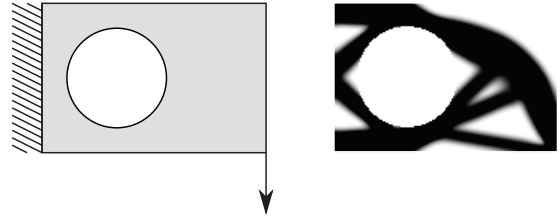

Fig. 7 The design domain, boundary conditions, and external load for the optimization of a cantilever beam with a fixed hole (left) and the optimized design obtained with a variant of the 88 line code using sensitivity filtering (right).

The load vector (line 19) and the support conditions (line 21) in this example are defined in the same way as in subsection 5.1. In order to distinguish between active and passive elements, a $\mathtt{nely} \times \mathtt{nelx}$ matrix `passive` is defined with 0 at elements free to change, 1 at elements fixed to be void, and 2 at elements fixed to be solid:

```
passive = zeros(nely,nelx);
for i = 1:nelx
  for j = 1:nely
    if sqrt((j-nely/2)^2+(i-nelx/3)^2) < nely/3
      passive(j,i) = 1;
    end
  end
end
```

These lines must be inserted in the 88 line code before the start of the optimization loop. The optimality criteria method must be modified by adding the following code between lines 78 and 79:

```
xPhys(passive==1) = 0;
xPhys(passive==2) = 1;
```

With these modifications, the 88 line code can be used to generate the optimized design shown in figure 7 by means of the following function call:

```
top88(150,100,0.5,3,5,1)
```

5.4 Heaviside projection filter

This subsection focuses on the implementation of a black-and-white projection filter. As an example, the implementation of the filter proposed by Guest et al (2004) is explained. This filter is referred to as the Heaviside projection filter in the present paper. The aim of the Heaviside projection filter is (1) to achieve a minimum length scale in the optimized design, and (2) to obtain black-and-white solutions. Guest et al (2004) apply this filter using

nodal design variables, but as shown by Sigmund (2007), it is equally applicable when element design variables are used (which is the case in the present paper).

The Heaviside filter is a modification of the original density filter (9) with a Heaviside step function that projects the density $\tilde{x}_e$ (from now on called the intermediate density) to a physical density $\bar{x}_e$. The physical density $\bar{x}_e$ equals one if $\tilde{x}_e > 0$ and zero if $\tilde{x}_e = 0$. In order to allow for the use of a gradient-based optimization scheme, the Heaviside function is replaced with the following smooth function:

$$\bar{x}_e = 1 - e^{-\beta\tilde{x}_e} + \tilde{x}_e e^{-\beta} \tag{21}$$

The parameter β controls the smoothness of the approximation: for β equal to zero, the Heaviside filter is identical to the original density filter; for β approaching infinity, the approximation approaches a true Heaviside step function. In order to avoid local minima and to ensure differentiability in the optimization, a continuation scheme is used where the parameter β is gradually increased from 1 to 512 by doubling its value every 50 iterations or when the change in terms of design variables between two consecutive designs becomes less than 0.01.

It should be noted that Guest et al (2004) include an extra term in equation (21) to ensure that the lower bound on the densities $\bar{x}_e$ is satisfied; this term is not necessary here due to the use of the modified SIMP approach (Sigmund, 2007).

The sensitivities of a function $f(\bar{x}_e)$ with respect to the intermediate densities $\tilde{x}_e$ are obtained by means of the chain rule:

$$\frac{\partial f}{\partial \tilde{x}_e} = \frac{\partial f}{\partial \bar{x}_e} \frac{\partial \bar{x}_e}{\partial \tilde{x}_e} \tag{22}$$

where the derivative of the physical density $\bar{x}_e$ with respect to the intermediate density $\tilde{x}_e$ is given by:

$$\frac{\partial \bar{x}_e}{\partial \tilde{x}_e} = \beta e^{-\beta\tilde{x}_e} + e^{-\beta} \tag{23}$$

The implementation of the Heaviside filter in the 88 line code as a third filter option (`ft = 3`) involves the following modifications.

First, the β parameter (`beta`) must be defined and the densities must be filtered before the start of the optimization loop. To this end, line 47 is replaced with the following lines:

```
beta = 1;
if ft == 1 || ft == 2
  xPhys = x;
elseif ft == 3
  xTilde = x;
  xPhys = 1-exp(-beta*xTilde)+xTilde*exp(-beta);
end
```

This code will lead to initial physical densities $\bar{x}_e$ that do not satisfy the volume constraint, which could be avoided by adjusting the initial values of the design variables x_e. In the present code, however, the optimality criteria update scheme is relied upon to correct the violation of the volume constraint.

Second, the modification of the sensitivities is accomplished by inserting the following supplementary `elseif` statement on line 68:

```
elseif ft == 3
  dx = beta*exp(-beta*xTilde)+exp(-beta);
  dc(:) = H*(dc(:).*dx(:)./Hs);
  dv(:) = H*(dv(:).*dx(:)./Hs);
```

Third, the application of the Heaviside filter to the densities is realized by means of the following additional `elseif` statement, to be inserted on line 78:

```
elseif ft == 3
  xTilde(:) = (H*xnew(:))./Hs;
  xPhys = 1-exp(-beta*xTilde)+xTilde*exp(-beta);
```

Finally, the continuation scheme for the regularization parameter β is implemented by inserting the following block of code at the end of the optimization loop:

```
if ft == 3 && beta < 512 && ...
                (loopbeta >= 50 || change <= 0.01)
  beta = 2*beta;
  loopbeta = 0;
  change = 1;
  fprintf('Parameter beta increased to %g.\n',beta);
end
```

The additional counter `loopbeta` must be initialized and incremented in the same way as the existing counter `loop`.

The modified code is used to optimize the MBB beam. The same parameter values are used as in subsection 3.4, except for the filter radius, which is reduced to 0.03 times the width of the design domain. The motivation for this reduction is that the material resource constraint prohibits the transformation of the topology obtained in the initial phase of the continuation scheme (which is similar to the topology obtained in subsection 3.4) into a black-and-white design consisting of bars with a large thickness.

Figure 8 shows the optimized design obtained with the three meshes. The optimized design is almost perfectly black-and-white and does not exhibit structural details smaller than the filter radius $r_{\min}$. The Heaviside projection filter relies on the compact support of the classical filter function to impose length scale in the solid regions, and therefore the Heaviside projection cannot be directly applied with the PDE filter. It can be observed also that the minimum length scale imposed on the material distribution does not prevent the occurrence of very small holes. This can be avoided by using a more advanced filter such as the morphological close-open or open-close filter (Sigmund, 2007) or by following a robust approach in the formulation
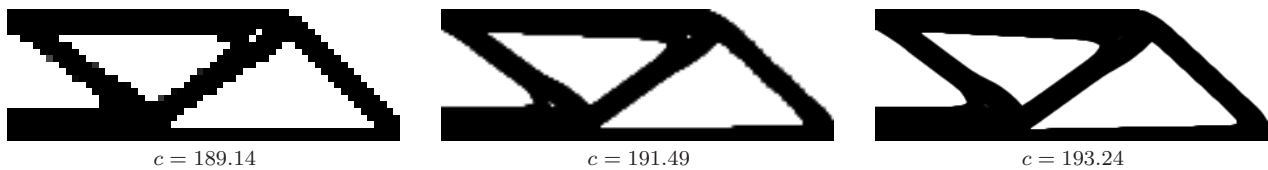
$c = 189.14$	$c = 191.49$	$c = 193.24$

Fig. 8 Optimized design of the MBB beam and corresponding compliance c obtained with the 88 line code extended by a Heaviside filter. A mesh with 60×20 elements (left), 150×50 elements (middle), and 300×100 elements (right) has been used.

of the optimization problem (Sigmund, 2009; Wang et al, 2010).

5.5 Other extensions

The implementation of additional extensions to the 88 line code should be relatively straightforward. The extension to three-dimensional problems may require a lot of modifications, but the general structure of the code would remain unchanged. Following the guidelines given by Bendsøe and Sigmund (2003), the 88 line code can be converted to a code for mechanism synthesis or for heat conduction problems. The optimality criteria based optimizer can be replaced with a more versatile optimization scheme, such as the method of moving asymptotes (MMA) introduced by Svanberg (1987), in order to enable the solution of problems with more than one constraint.

6 Performance

6.1 Computation time

In this subsection, the computation time for the 88 line code (and the variants presented sections 4 and 5.4) is compared with the original 99 line code. The MBB beam optimization problem introduced earlier is considered as a benchmark problem, using the same parameter values as in subsection 3.4.

Table 1 Computation time in seconds per iteration for the optimization of the MBB beam using sensitivity filtering.

Mesh size	60×20	150×50	300×100
99 line code	0.65	75.19	-
88 line code	0.15	0.72	1.85
CONV2 based filtering code	0.13	0.69	1.98
PDE based filtering code	0.13	0.78	2.18

Table 1 gives an overview of the computation time in seconds per iteration for the optimization of the MBB beam using sensitivity filtering. Results are given for four variants of the optimization code and for three different mesh sizes. The computation times have been determined as the average over the first ten iterations of the optimization loop, using a Lenovo Thinkpad X301 laptop with an Intel Core2 Duo U9400 processor, 2 GB memory, Windows XP with SP3 (32-bit x86), and MATLAB R2010a. It is clear from the table that the new 88 line implementation is significantly faster than the original 99 line code. For the mesh with 150×50 elements, a factor of 100 speed improvement is accomplished. The 99 line code has not been tested using the mesh with 300×100 elements as the computation time becomes excessively large. The computation time for the alternative implementations using `conv2` based filtering and PDE based filtering is almost equal to the computation time for the 88 line code.

Table 2 Computation time in seconds per iteration for the optimization of the MBB beam using density filtering.

Mesh size	60×20	150×50	300×100
88 line code	0.12	0.94	5.67
CONV2 based filtering code	0.16	0.78	3.30
PDE based filtering code	0.19	1.79	10.08

Table 2 shows the results obtained with a density filter, using the same configuration as for the sensitivity filter. No results are given for the 99 line code as it does not include a density filter. The computation time is slightly higher than for the sensitivity filter, due to the application of the density filter in every iteration of the bisection algorithm used to determine the Lagrangian multiplier λ. This is especially true for the problem with the largest mesh, and for the code using PDE based filtering, where the application of the density filter involves a relatively costly backsubstitution operation. As the PDE based filter is volume-preserving, this could be avoided using the design variables instead of the physical densities to check the volume constraint in the bisection algorithm. Moreover, the computational cost can be significantly reduced by employing an iterative solver (Lazarov and Sigmund, 2010).

Finally, the performance of the code extended by a Heaviside filter is described in table 3. Compared to the standard density filtering code, the additional Heaviside projection has no significant impact on the computation time. The computation time per iteration is slightly lower due to the use of a smaller filter radius $r_{\min}$ in the

Table 3 Computation time in seconds per iteration for the optimization of the MBB beam using Heaviside filtering.

Mesh size	60×20	150×50	300×100
Code with Heaviside filter	0.13	0.86	4.65

example problem, which leads to a sparser coefficient matrix H, reducing the number of operations required to multiply the coefficient matrix H with the design variables or the sensitivities. It should be noted, however, that the use of the Heaviside filter requires the application of a continuation scheme, which implies that the number of iterations becomes considerably larger.

6.2 Memory usage

While the use of the `sparse` function to assemble the stiffness matrix leads to a vast improvement in terms of computation time, it also increases the program's memory footprint. The index vectors `iK` and `jK` and the vector `sK` with non-zero entries are relatively large and remain in memory throughout the entire optimization loop. Each of these vectors has the size of the element stiffness matrix times the number of elements, which is considerably more than the size of the stiffness matrix itself. Moreover, the `sparse` function requires the index vectors `iK` and `jK` to be defined as double precision arrays, prohibiting the use of a more memory efficient integer type. In contrast to the `sparse` function built-in in MATLAB, the `sparse2` function from CHOLMOD does accept integer type index vectors.

In order to get a rough idea of the memory requirements for the 88 line code and its variants, an informal test has been conducted: the example problem from the previous subsection has been solved multiple times, each time incrementing the mesh size, until the computer ran out of memory. The same computer has been used as for the determination of the computation time. The test has been performed using both sensitivity and density filtering, leading to identical results.

The largest problem that could be solved with the 88 line code consisted of $300 \times 100 = 30000$ elements. The code using `conv2` based filtering requires less memory as it avoids the definition of the (sparse but nonetheless large) coefficient matrix H, using a small matrix h instead that represents the non-zero part of the filter kernel. As a consequence, this code allowed for the solution of a problem with $700 \times 233 = 163100$ elements. The code based on PDE filtering performs in between and allowed us to solve a problem with $600 \times 200 = 120000$ elements. The extension of the 88 line code with a Heaviside filter has no noticeable influence on memory usage; however, due to the use of a smaller filter radius $r_{\min}$ in the example problem, the coefficient matrix H becomes sparser, and a problem with $350 \times 117 = 40950$ elements could be solved. The original 99 line code has not been tested, as the computation time becomes prohibitively large for problems of these dimensions.

7 Conclusion

This paper presents a MATLAB code for topology optimization. The code is considered as the successor to the 99 line code presented by Sigmund (2001). It is published with the same objective: to provide students and newcomers to the field with a very simple implementation of a topology optimization algorithm that can serve as an introductory example and as a basis for further developments. The code can be download from the web site www.topopt.dtu.dk.

The major difference with respect to the original 99 line code is the computational efficiency. An improvement in speed with a factor of 100 has been measured for an example problem with 7500 elements. This has mainly been accomplished by means of loop vectorization and memory preallocation.

In addition, the code has been extended by a density filter. The inclusion of a density filter has an important educational value, as it paves the road for the implementation of more sophisticated filters such as the Heaviside filter also discussed in the paper.

Special care has been taken not to compromise the simplicity of the code. As a result, the new code is characterized by the same readability as the original 99 line code, although the number of lines has been reduced to 88.

The paper also presents two alternative implementations. The first alternative takes advantage of the `conv2` function built-in in MATLAB to filter densities and sensitivities, so reducing the number of lines to 71 without affecting the computational cost or the readability of the code (for those familiar with the `conv2` function). The second alternative uses a filter based on a Helmholtz type differential equation, allowing for the use of a finite element solver to perform the filtering operation. This is beneficial for problems with a complex geometry or when the optimization problem is solved in parallel.

References

Alberty J, Carstensen C, Funken S (1999) Remarks around 50 lines of Matlab: short finite element implementation. Numerical Algorithms 20(2–3):117–137

Alberty J, Carstensen C, Funken S, Klose R (2002) Matlab implementation of the finite element method in elasticity. Computing 69(3):239–263

Allaire G (2009) Shape and topology optimization by the level set method. URL http://www.cmap.polytechnique.fr/~allaire

Bendsøe M, Sigmund O (2003) Topology Optimization. Theory, Methods and Applications. Springer

Bendsøe M (1989) Optimal shape design as a material distribution problem. Structural optimization 1:193–202

Bourdin B (2001) Filters in topology optimization. International Journal for Numerical Methods in Engineering 50(9):2143–2158

Bruns TE, Tortorelli DA (2001) Topology optimization of non-linear elastic structures and compliant mechanisms. Computer Methods in Applied Mechanics and Engineering 190(26-27):3443–3459

Challis VJ (2010) A discrete level-set topology optimization code written in Matlab. STRUCTURAL AND MULTIDISCIPLINARY OPTIMIZATION 41(3):453–464

Dabrowski M, Krotkiewski M, Schmid D (2008) MILAMIN: MATLAB-based finite element method solver for large problems. Geochemistry Geophysics Geosystems 9(4)

Davis T (2007) Creating sparse finite-element matrices in MATLAB. Guest blog in Loren on the Art of MATLAB, http://blogs.mathworks.com/loren/2007/03/01/creating-sparse-finite-element-matrices-in-matlab/. URL http://blogs.mathworks.com/loren/

Davis T (2008) User Guide for CHOLMOD: a sparse Cholesky factorization and modification package. Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Díaz A, Sigmund O (1995) Checkerboard patterns in layout optimization. Structural Optimization 10(1):40–45

Guest J, Prevost J, Belytschko T (2004) Achieving minimum length scale in topology optimization using nodal design variables and projection functions. International Journal for Numerical Methods in Engineering 61(2):238–254

Jog C, Haber R (1996) Stability of finite element models for distributed-parameter optimization and topology design. Computer Methods in Applied Mechanics and Engineering 130(3–4):203–226

Lazarov B, Sigmund O (2009) Sensitivity filters in topology optimisation as a solution to helmholtz type differential equation. In: In Proc. of the 8th World Congress on Structural and Multidisciplinary Optimization

Lazarov B, Sigmund O (2010) Filters in topology optimization based on Helmholtz type differential equations (submitted for publication)

Sigmund O (1994) Design of material structures using topology optimization. PhD thesis, DCAMM S-report S69, Department of Solid Mechanics, Technical University of Denmark

Sigmund O (1997) On the design of compliant mechanisms using topology optimization. Mechanics of Structures and Machines 25(4):493–524

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Structural and Multidisciplinary Optimization 21(2):120–127

Sigmund O (2007) Morphology-based black and white filters for topology optimization. Structural and Multidisciplinary Optimization 33(4-5):401–424

Sigmund O (2009) Manufacturing tolerant topology optimization. Acta Mechanica Sinica 25(2):227–239

Sigmund O, Petersson J (1998) Numerical instabilities in topology optimization: A survey on procedures dealing with checkerboards, mesh-dependencies and local minima. Structural Optimization 16(1):68–75

Suresh K (2010) A 199-line Matlab code for Pareto-optimal tracing in topology optimization. Structural and Multidisciplinary Optimization Published online

Svanberg K (1987) Method of moving asymptotes - a new method for structural optimization. International Journal for Numerical Methods in Engineering 24(2):359–373

The MathWorks (2010) MATLAB Programming Fundamentals

Wang F, Lazarov B, Sigmund O (2010) On projection methods, convergence and robust formulations in topology optimization (submitted for publication)

Zhou M, Rozvany G (1991) The COC algorithm, part II: Topological, geometrical and generalized shape optimization. Computer Methods in Applied Mechanics and Engineering 89(1–3):309–336

Appendix - MATLAB code

```matlab
1  %%%% AN 88 LINE TOPOLOGY OPTIMIZATION CODE %%%%
2  function top88(nelx,nely,volfrac,penal,rmin,ft)
3  %% MATERIAL PROPERTIES
4  E0 = 1;
5  Emin = 1e-9;
6  nu = 0.3;
7  %% PREPARE FINITE ELEMENT ANALYSIS
8  A11 = [12  3 -6 -3;  3 12  3  0; -6  3 12 -3; -3  0 -3 12];
9  A12 = [-6 -3  0  3; -3 -6 -3 -6;  0 -3 -6  3;  3 -6  3 -6];
10 B11 = [-4  3 -2  9;  3 -4 -9  4; -2 -9 -4 -3;  9  4 -3 -4];
11 B12 = [ 2 -3  4 -9; -3  2  9 -2;  4  9  2  3; -9 -2  3  2];
12 KE = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11]);
13 nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx);
14 edofVec = reshape(2*nodenrs(1:end-1,1:end-1)+1,nelx*nely,1);
15 edofMat = repmat(edofVec,1,8)+repmat([0 1 2*nely+[2 3 0 1] -2 -1],nelx*nely,1);
16 iK = reshape(kron(edofMat,ones(8,1))',64*nelx*nely,1);
17 jK = reshape(kron(edofMat,ones(1,8))',64*nelx*nely,1);
18 % DEFINE LOADS AND SUPPORTS (HALF MBB-BEAM)
19 F = sparse(2,1,-1,2*(nely+1)*(nelx+1),1);
20 U = zeros(2*(nely+1)*(nelx+1),1);
21 fixeddofs = union([1:2:2*(nely+1)],[2*(nelx+1)*(nely+1)]);
22 alldofs = [1:2*(nely+1)*(nelx+1)];
23 freedofs = setdiff(alldofs,fixeddofs);
24 %% PREPARE FILTER
25 iH = ones(nelx*nely*(2*(ceil(rmin)-1)+1)^2,1);
26 jH = ones(size(iH));
27 sH = zeros(size(iH));
28 k = 0;
29 for i1 = 1:nelx
30   for j1 = 1:nely
31     e1 = (i1-1)*nely+j1;
32     for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
33       for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
34         e2 = (i2-1)*nely+j2;
35         k = k+1;
36         iH(k) = e1;
37         jH(k) = e2;
38         sH(k) = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2));
39       end
40     end
41   end
42 end
43 H = sparse(iH,jH,sH);
44 Hs = sum(H,2);
45 %% INITIALIZE ITERATION
46 x = repmat(volfrac,nely,nelx);
47 xPhys = x;
48 loop = 0;
49 change = 1;
50 %% START ITERATION
51 while change > 0.01
52   loop = loop + 1;
53   %% FE-ANALYSIS
54   sK = reshape(KE(:)*(Emin+xPhys(:)'.^penal*(E0-Emin)),64*nelx*nely,1);
55   K = sparse(iK,jK,sK); K = (K+K')/2;
56   U(freedofs) = K(freedofs,freedofs)\F(freedofs);
57   %% OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
58   ce = reshape(sum((U(edofMat)*KE).*U(edofMat),2),nely,nelx);
59   c = sum(sum((Emin+xPhys.^penal*(E0-Emin)).*ce));
60   dc = -penal*(E0-Emin)*xPhys.^(penal-1).*ce;
61   dv = ones(nely,nelx);
62   %% FILTERING/MODIFICATION OF SENSITIVITIES
63   if ft == 1
64     dc(:) = H*(x(:).*dc(:))./Hs./max(1e-3,x(:));
65   elseif ft == 2
66     dc(:) = H*(dc(:)./Hs);
67     dv(:) = H*(dv(:)./Hs);
68   end
69   %% OPTIMALITY CRITERIA UPDATE OF DESIGN VARIABLES AND PHYSICAL DENSITIES
```

```
70    l1 = 0; l2 = 1e9; move = 0.2;
71    while (l2-l1)/(l1+l2) > 1e-3
72      lmid = 0.5*(l2+l1);
73      xnew = max(0,max(x-move,min(1,min(x+move,x.*sqrt(-dc./dv/lmid)))));
74      if ft == 1
75        xPhys = xnew;
76      elseif ft == 2
77        xPhys(:) = (H*xnew(:))./Hs;
78      end
79      if sum(xPhys(:)) > volfrac*nelx*nely, l1 = lmid; else l2 = lmid; end
80    end
81    change = max(abs(xnew(:)-x(:)));
82    x = xnew;
83    %% PRINT RESULTS
84    fprintf(' It.:%5i Obj.:%11.4f Vol.:%7.3f ch.:%7.3f\n',loop,c, ...
85      mean(xPhys(:)),change);
86    %% PLOT DENSITIES
87    colormap(gray); imagesc(1-xPhys); caxis([0 1]); axis equal; axis off; drawnow;
88  end
```